1 XAIF Schema Overview

In this section we present the syntax of the XAIF in more detail. Figure 1 shows the UML [1] model for the XAIF schema, using the approach described in [2, 3]. Because of space considerations, the model does not contain some elements or relationships. For a full version of the current schema draft, refer to www.mcs.anl.gov/xaif.

The XAIF representation consists of a series of nested graphs. All graph elements are of type `GraphType`, whose definition is similar to the XGMML notion of a graph [4]. All elements of `GraphType` contain at least one element of `VertexType` and zero or more elements of `EdgeType`. All elements of `VertexType` have identifiers that are unique within the parent graph element. Edges have unique identifiers, as well as key references to source and target vertices. For clarity, some relationships have been omitted. In general, if a UML class name ends with "Graph", the corresponding schema type inherits from `GraphType`. Similarly, types with names ending with "Vertex" or "Edge" extend `VertexType` and `EdgeType`, respectively. We have shown the complete details for the graph-vertex-edge relationships for the `CallGraph` elements.

At the highest level, the program is represented by a `CallGraph` element, whose children are vertices corresponding to subroutines and edges signifying subroutine calls.

`CallGraph`, `ControlFlowGraph`, `BasicBlockGraph`, `BasicScopeGraph`, and all statement graph elements can contain optional `Properties` elements encapsulated in a property tree named after the corresponding graph. These graphs may also include a `SymbolTable` element (described in more detail later) for storing descriptive information about variable, constant, and subroutine symbols used within each scope.

Each `CallGraph` vertex contains a `ControlFlowGraph` element, whose vertices and edges represent the control flow of the program. A `ControlFlowVertex` can contain a `BasicBlockGraph`, a `ForLoopGraph`, an `IfConditionGraph`, or, in general, any statement that affects the flow of control in the computation.

Each `ControlFlowVertex` can contain either a `BasicBlockGraph` or a graph corresponding to a compound statement (e.g., a `ForLoopGraph`). The portions of the code that are actually augmented with derivative computations are contained within `BasicBlockGraphs`, which correspond to basic blocks in the code. A vertex of a `BasicBlockGraph` can be a `BasicScopeGraph` (used to represent scoping within a basic block), a `SubroutineCallGraph`, or an `AssignmentStatementGraph`.

Only the assignment statements containing active variables (or loop indices) are included in the XAIF as `AssignmentStatementGraphs`. The left-hand side of an assignment vertex is limited to a `VariableReferenceVertex`, while the right-hand side can be a `ConstantVertex`, a `VariableReferenceVertex`, or an `ExpressionGraph`. The representation of expressions in the `ExpressionGraph` is straightforward, including both Boolean and arithmetic operators. We used a substitution group for the different kinds of expression graph vertices, i.e., each of the members of the group can be a child of the `ExpressionGraph` element.

Transformation tools operate at different granularities of the graph hierarchy. For example, a forward-mode module using statement-level reverse mode needs access only to the XAIF for assignment statements. Other modules may implement strategies that require basic block-level XAIF, while some reverse-mode tools may need access to control flow or call graph information. The XAIF is flexible enough to allow the independent processing of different levels of the graph hierarchy.

All variable and constant reference vertices contain a required `symbolId` attribute and an optional `symbolTableId` attribute. The unique combination of these identifiers can be used to access information about the variable or constant in the corresponding symbol table. As mentioned earlier, the `SymbolTable` element can be included at many different levels, allowing for flexibility when generating XAIF for processing at different levels. For example, an existing first-order differentiation module operates only at the assignment statement level and requires symbol information for the symbols in each statement.

In the XAIF, a symbol table can be attached to each assignment statement. On the other hand, if a module operates on the basic block level, such replication of symbol information can be avoided by including a single symbol table for each basic block. Entries in the symbol (`Symbol` elements) contain several fields, such as the type and shape of a variable. The information stored for each symbol can easily be extended with new fields.

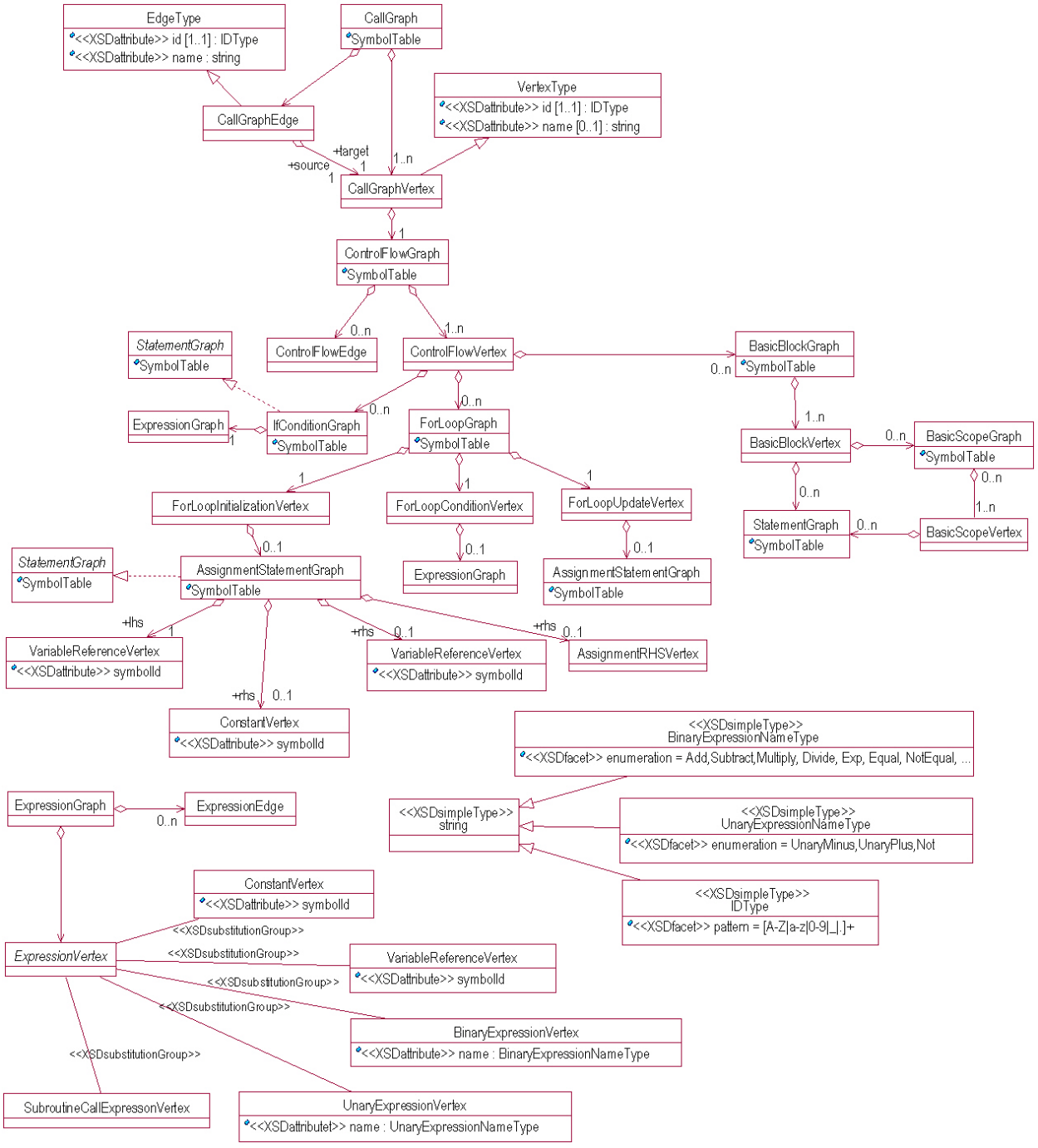

Figure 1. XAIF Schema Model

References

[1] BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[2] CARLSON, D. Modeling XML vocabularies with UML: Part III. http://www.xml.com/lpt/a/2001/-10/10/uml.html.

[3] CARLSON, D. *Modeling XML Applications with UML: Practical e-Business Applications*. Addison-Wesley, 2001.

[4] PUNIN, J., AND KRISHNAMOORTHY, M. Extensible Markup and Modeling Language (XGMML) draft specification in the XML.org XML Standards Report. http://www.zapthink.com/report.html, 2001.